

Tamer Dallou, Ben Juurlink

Hardware-based task dependency resolution for the StarSs programming model

Conference object, Postprint version

This version is available at <http://dx.doi.org/10.14279/depositonce-5781>.



Suggested Citation

Dallou, Tamer; Juurlink, Ben: Hardware-based task dependency resolution for the StarSs programming model. - In: 2012 41st International Conference on Parallel Processing Workshops : ICPPW. - New York, NY [u.a.] : IEEE, 2012. - ISBN: 978-1-4673-2509-7. - pp. 367-374. - DOI: 10.1109/ICPPW.2012.53.
(Postprint version is cited, page numbers differ.)

Terms of Use

© © 2012 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

Hardware-Based Task Dependency Resolution for the StarSs Programming Model

Tamer Dallou and Ben Juurlink
Embedded Systems Architecture Group
Technische Universität Berlin
Einsteinufer 17, 10587 Berlin, Germany
{dallou, b.juurlink}@tu-berlin.de

Abstract—Recently, several programming models have been proposed that try to relieve parallel programming. One of these programming models is StarSs. In StarSs, the programmer has to identify pieces of code that can be executed as tasks, as well as their inputs and outputs. Thereafter, the runtime system (RTS) determines the dependencies between tasks and schedules ready tasks onto worker cores. Previous work has shown, however, that the StarSs RTS may constitute a bottleneck that limits the scalability of the system and proposed a hardware task management system called Nexus to eliminate this bottleneck. Nexus has several limitations, however. For example, the number of inputs and outputs of each task is limited to a fixed constant and Nexus does not support double buffering. In this paper we present Nexus++ that addresses these as well as other limitations. Experimental results show that double buffering achieves a speedup of $54\times/143\times$ with/without modeling memory contention respectively, and that Nexus++ significantly enhances the scalability of applications parallelized using StarSs.

I. INTRODUCTION

Due to the advent of multicore architectures, several parallel programming models have been proposed that aim at relieving parallel programming. Examples include Google's MapReduce [4], Intel's TBB [14], and StarSs [12]. StarSs, like OpenMP [3], enables the programmer to express parallelism by adding pragmas to the code. These pragmas identify pieces of code that can be executed as *tasks*, as well as their *inputs* and *outputs*. Based on the inputs and outputs, the RTS can determine the dependencies between tasks and schedule ready tasks onto cores that execute the tasks. The programmer, therefore, does not have to explicitly express dependencies between tasks and the corresponding synchronization. Furthermore, the RTS can also transparently optimize data reuse between tasks and coarsen tasks, thereby relieving the programmer from these burdens.

Previous work [10] has shown, however, that the StarSs RTS, when implemented in software, can be a bottleneck that limits the scalability of applications parallelized using StarSs. Roughly speaking, the RTS cannot compute task dependencies and attend to finished tasks fast enough to keep all *worker cores* that execute the tasks busy. The same work therefore proposed a hardware task management system called *Nexus* to accelerate the RTS. In Nexus, task dependencies are computed using hardware hash tables and a scalable synchronization mechanism with the worker cores is provided. Results show that Nexus improves the scalability of a synthetic application modeled after H.264 decoding by a factor of 4.3 when using 16 worker cores.

Even though Nexus improves the scalability significantly, it has several limitations. For example, since the hash table entries have a fixed size, the number of inputs and outputs

of each task is limited (up to 5 in [10], [9]). Similarly, the number of tasks that can depend on a certain data segment is limited. This limits the applicability of Nexus, i.e., not all StarSs applications can be executed on a multicore system with Nexus. Another limitation is that Nexus does not support double buffering, which allows executing one task while fetching the input data of another task. In [10] double buffering was not needed because the data transfer time was negligible.

In this paper we present Nexus++ that addresses these as well as other limitations. Nexus++ main contributions include: first, it solves the constraint on the maximum number of inputs/outputs a task can have by introducing *dummy* tasks. It also solves the constraint on the dependency count of a certain data segment by adding dummy entries to the list of tasks that depend on this data segment. Second, it support double (in fact arbitrary) buffering by providing a *Task Controller* at each worker core that buffers tasks before they are executed. Third, it implements task dependency resolution more efficiently, since fewer resources and computations are needed. Fourth, Nexus++ implementation is platform-independent, since its parameters are fully configurable, while Nexus was integrated in a simulator of the Cell processor.

A SystemC model has been developed to validate and evaluate the design. The preliminary results show that double buffering achieves a speedup of $54\times$ for up to 64 cores. For 128 cores and more, the speedup gain starts to decrease because the master core that generates tasks and submits them to Nexus++ cannot generate them fast enough to keep all worker cores busy, and due to limited memory bandwidth. The results also show that applications that could not be executed by Nexus, such as Gaussian elimination with partial pivoting, which resembles the LINPACK benchmark, can be executed efficiently on a multicore system with Nexus++.

This paper is organized as follows. Overview of the StarSs programming model and related work are described in Section II. Nexus++ and its features are described in Section III. In Section IV the simulation environment and the employed benchmarks are described. The experimental results are presented in Section V, and conclusions are drawn in Section VI.

II. BACKGROUND

A. StarSs

StarSs is a task-based programming model, which enables exploitation of task-level parallelism, regardless of the target architecture. StarSs provides programmers with *pragmas*, an annotations added to the serial code, to identify potential pieces of code that can run in parallel. The programmer does not need to care about synchronization between tasks, as this is done implicitly by the StarSs RTS. Listing 1 shows an example of exploiting parallelism using pragmas.

The example in Listing 1 shows that function *decode()* is called inside a nested loop, processing the elements of matrix

X. Calculating *decode()* for each element requires the values of the left and up-right cells. This example represents macroblock wavefront decoding in H.264 [16], for one 1920×1088 frame in blocks of 16×16 , and it is one of the benchmarks used to evaluate Nexus++.

```
int* X[120][68];
#pragma css task input(left[16][16],\
upright[16][16]) inout(this[16][16])
void decode(int* left, int* upright, int* this){...}
void main(){
    int i, j;
    init_matrix(X);
    for(i=0; i<120; i++)
        for(j=0; j<68; j++)
            decode(X[i][j-1], X[i-1][j+1], X[i][j]);
    #pragma css barrier
}
```

Listing 1. StarSs example of macroblock wavefront decoding in H.264

Annotating a function with the *css task* pragma defines a task. The inputs/outputs of the task should also be specified as with function *decode()* in Listing 1. StarSs also provides several synchronization pragmas such as the *css barrier* pragma.

A source-to-source compiler transforms the annotated function calls to runtime library calls, which generate a task out of each function call, and add it to the task graph. As in the example of Listing 1, every time function *decode()* is called, a task is generated.

Having identified the tasks and the direction of their parameters, the StarSs environment builds, at run time, the task graph, and the task-level parallelism is detected and exploited.

B. Related Work

Several hardware scheduling units have been proposed in literature. Most of them, however, assume independent tasks and are optimized for a certain application, a certain platform, or both. For example, Carbon [7] assumes independent tasks and uses hardware queues to retrieve tasks with low latency.

In StarSs, tasks can be dependent and it is the responsibility of the RTS to determine their dependencies. An example of a hardware accelerator targeted at a certain application domain is a hardware task scheduler optimized for H.264 decoding [1]. It requires, however, that the programmer specifies the dependencies between blocks. Etsion et al. [6] also proposed a hardware task management unit for the StarSs RTS, based on the similarity between task dependency checking and the instruction scheduler of an out-of-order processor. It was evaluated using high-level simulations, however, and detailed hardware models were not developed.

As mentioned before, our work builds upon Nexus [10], which was integrated in a simulator of the Cell processor.

III. NEXUS++ HARDWARE TASK MANAGEMENT SYSTEM

The multicore system under consideration, shown in Figure 1, is assumed to have one *Master Core* that executes the main thread and creates *Task Descriptors*, and several worker cores that execute the tasks. A *Task Descriptor* contains task-related information such as its function pointer and input/output list. Nexus++ is responsible for the task graph management usually carried out by the software RTS. In an *n*-core system (one master core and $(n - 1)$ worker cores), Nexus++ is composed of *n* hardware modules:

- one *Task Maestro*, which is mainly responsible for dependency resolution, task scheduling, and load balancing,
- and $n - 1$ local *Task Controllers* (TCs), one per worker core, and are mainly responsible for task buffering.

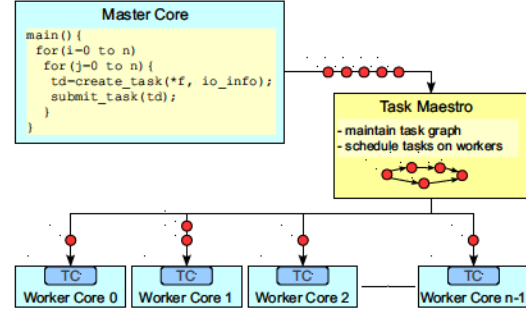


Fig. 1. Nexus++ in a multicore system

A. System Description

The different components of Nexus++ shown in Figure 2 are described through explaining a task's life cycle.

Tasks submission: When the *Master Core* executes the main program, it generates the *Task Descriptors* and sends them to the *Task Maestro* via the *Get TDs* block; the first hardware block of the *Task Maestro*; which communicates with the *Master Core* and receives variable-length *Task Descriptors* (depending on the number of inputs/outputs per task) and writes them to the *TDs Buffer*. This block is important so that the *Master Core* is not blocked while the *Task Maestro* is busy processing an earlier submitted task. It also enables direct communication between the master core and the *Task Maestro*, avoiding off-chip communication overhead, which is one of the scalability limiting factors of Nexus[9]. After having received a *Task Descriptor*, the *Get TDs* block writes its size to a FIFO list called the *TDs Sizes* list. If this list is full, the *Master Core* stalls and stops sending new *Task Descriptors*.

Storing tasks: Once the *TDs Sizes* list is written, it triggers the *Write TP* block, which reads the size of the recently received *Task Descriptor* from the *TDs Sizes* list, then it reads the *Task Descriptor* from the *TDs Buffer*, appends some meta data to it, and finally writes it to the main task storage table in Nexus++ which is called the *Task Pool*. The full format of a *Task Descriptor* in the *Task Pool* is shown in Table I. The 1st column in Table I is the index at which tasks are stored. This index is determined by the *Write TP* block, which reads the *TP Free indices* list, that stores initially all indices of the *Task Pool*. After the completion of a task, its *Task Pool*'s index is written back to the *TP Free indices* list.

Inside Nexus++, a task is identified by its *Task Pool* index. This is important to directly address a specific entry in the table, rather than searching the table for that entry.

The *busy* column of a *Task Descriptor* is a boolean flag indicating whether this *Task Descriptor* is currently under processing by one of the blocks of the *Task Maestro* or not. This is to ensure exclusive access to any entry in the *Task Pool* at a certain time, and hence, preventing dead locks.

The **f* column of a *Task Descriptor* indicates the function pointer of that task. The *DC* column stands for *Dependence Counter*, which records how many dependencies must be fulfilled before this task can be scheduled to run, i.e., how many inputs of this task are outputs of older tasks.

The *nD* stores the *number of dummy entries* that are linked to this *Task Descriptor*. Adding dummy entries to the *Task Pool* is the mechanism used to overcome the limit on the number of inputs/outputs a task can have. This mechanism is explained in Section III-C.

Columns *nP* and the following ones indicate the number

addr	busy	tp_i	*f	DC	nD	nP	P_1	P_2	...	P_8 or ptr_next	Dummy
17	0	17	0xABCD	0	0	8	1A/4/in	2A/4/in	...	1B/4/out	
98	0	98	0xDCBA	1	1	10	1B/4/in	2B/4/inout	...	99/...	
99	0	99	-	-	-	-	8B/4/in	9B/4/out	10B/4/out	-	

TABLE I

THE TASK POOL. (TP_I: TP INDEX, *F: FUNC. PTR, DC: DEPENDENCE COUNT, ND: NUM. DUMMY ENTRIES, NP: NUM. PARAMETERS, P_x : PARAMETER $_x$)

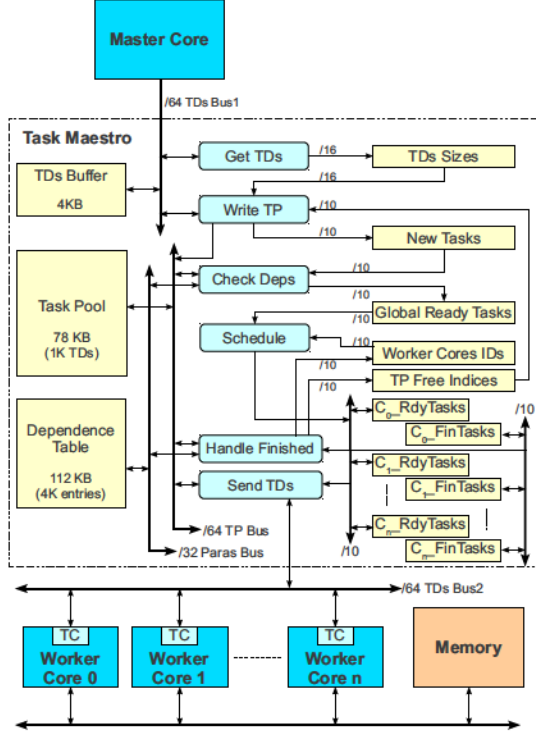


Fig. 2. Nexus++ modules block diagram

of inputs/outputs, and their information, respectively. An input/output of a task is stored in the format: (*base memory address, size, and access mode*), where the access mode can be either input, output, or inout.

Resolving tasks dependencies: Once the *Write TP* block has finished storing a task in the *Task Pool*, it writes this task's *ID* (*Task Pool's* index) in a FIFO list called *New Tasks*, the event that triggers the *Check Deps* block. The latter block is responsible for checking whether the new submitted task is ready or not, by checking the new submitted task inputs/outputs against all those of the previously submitted tasks. The task dependence graph is stored inside the *Dependence Table*. The process of dependency resolution is described in detail in Section III-B to emphasize on its capabilities and efficiency.

Scheduling tasks: Once a ready task is found by the *Check Deps* block, it writes its *ID* to a FIFO list called the *Global Ready Tasks* list. This event triggers the *Schedule* block, which is responsible for scheduling ready tasks onto the worker cores. Another FIFO list called the *Worker Cores IDs* list contains initially all worker cores IDs (repeated "buffering depth" times). The *Schedule* block reads the latter FIFO for a worker core ID and schedules the last found ready task on this worker core. This simple round-robin scheduling mechanism achieves load balancing between cores, since whenever a core finishes running a task, the core's ID is written back at the tail of the *Worker Cores IDs* list.

The *Task Maestro* has two FIFO lists for each worker core. The first one called the *C_iRdyTasks* (*Core_i Ready Tasks*) list,

and the second one is the *C_iFinTasks* (*Core_i Finished Tasks*) list. Scheduling a task on a core is done by writing the task's *ID* in that core's *C_iRdyTasks* list. *C_iFinTasks* lists are used later upon completion of tasks.

Send ready tasks to worker cores: Once the *RdyTasks* list of a certain core is written, this 1-bit *list_written_event* is communicated to the corresponding worker core. In each of the worker cores, a small and simple unit called the local *Task Controller* (TC) is integrated. The *Task Controller* is mainly responsible for communication with the *Task Maestro*, and to enable buffering of tasks.

A *Task Controller* contains four pipelined hardware blocks, namely the *Get TD*, *Get Inputs*, *Run Task*, and *Put Outputs* blocks. The first of them is the *Get TD* block, which is triggered upon writing a new task ID to the corresponding core's *RdyTasks* list. The *Get TD* block is responsible for fetching parts (*f and input/output list) of the *Task Descriptors* from the *Task Maestro*. This is done by sending a 1-bit request signal to the *Task Maestro*; the event that is handled by the *Send TDs* block in the *Task Maestro*. The latter block works in a round-robin fashion. It checks all the requests from the different *Task Controllers*, and whenever it finds an active one, it reads the *RdyTasks* list corresponding to the incoming active signal and gets the ready task *ID*. Since a task *ID* is the index at which it is stored in the *Task Pool*, the *Send TDs* block reads the *Task Descriptor* at that index directly without searching the *Task Pool*. After the *Send TDs* block have sent the requested *Task Descriptor* to the requesting worker core, it writes the sent task *ID* to that core's *FinTasks* list, which is important upon task completion as will be shown later.

Sending tasks to worker cores upon requests from the local *Task Controllers* insures that the *Send TDs* block in the *Task Maestro* will not waste any clock cycle waiting for a local *Task Controller*, due to for example a handshaking protocol or full buffer at that local *Task Controller*.

Run tasks: After getting a task from the *Task Maestro*, the *Get Inputs* block at the *Task Controller* side, prefetches the task code and inputs from memory. Then, the *Run Task* block passes the task to the worker core to run it, and finally the *Put Outputs* block writes the outputs back to memory, and notifies the *Task Maestro*, via a 1-bit notification signal, of task completion.

Finalize tasks, and update the task graph: The *task-finished* notification signals from the local *Task Controllers* are handled by the *Handle Finished* block in the *Task Maestro*. The latter block also works in a round-robin fashion; it continuously checks the notification signals from the different cores, and whenever it finds an active one, it performs two things: first, it acknowledges the corresponding local *Task Controller* of the observation of its *task-finished* signal, so the local *Task Controller* deactivates its *task-finished* signal consequently.

The second thing the *Handle Finished* block performs is that it reads the *FinTasks* list of the corresponding worker core. The value read is the *ID* of the finished task, since the *FinTasks* list was written by the *Send TDs* block immediately after having sent the *Task Descriptor* to the corresponding worker core. After reading the finished task *ID*, the *Handle Finished* block reads the input/output list of the finished task

from the *Task Pool*, updates the *Dependence Table* and kicks-off pending tasks, if any. Finally, the *Handle Finished* block deletes the task from the *Task Pool*, adds the task *ID* to the *TP Free indices* list, and adds the worker core *ID* to the *Worker Cores IDs* list.

Since inside Nexus++ any task is identified by the index at which its *Task Descriptor* is stored in the *Task Pool*, the size and access time of the different tables and FIFO lists are reduced. Furthermore all events and notifications are one-bit signals, which ensures low communication overhead between the *Task Maestro* blocks, the *Task Controller* blocks, and of course between the *Task Maestro* and the *Task Controllers*.

Both Nexus and Nexus++ provide dependency resolution. However, Nexus can only deal with tasks with a limited number of inputs/outputs. It also can deal with dependency patterns where only few, limited number of tasks depend on a certain task. In addition, Nexus proposed TCs, but did not implement them. Nexus++ solves the above limitations as described next.

B. Dependency Resolution

Dependency resolution between tasks is accomplished inside the *Task Maestro* by the *Check Deps* block, *Handle Finished* block, and the *Dependence Table* along with the *Dependence Counter* associated with every *Task Descriptor* in the *Task Pool*. Currently, dependencies between tasks are decided by comparing the base addresses of the inputs/outputs of the different tasks.

The Dependence Table: The place where dependence information are stored. Each input/output that is accessed by a task will have an entry in the *Dependence Table* indicating its access mode, and a *Kick-Off List* that contains the *IDs* of tasks waiting for this address to be produced before they can run.

The *Dependence Table* is a hash table with a simple separate chaining hash collisions resolution algorithm $h()$. The different fields of it are shown in Table II. The first column *hAddr* is the hash address, followed by a valid bit in the *v* column, followed by the full memory address in the *fAddr* column. Size and access mode of this memory segment are stored in the *Size* and *isOut* columns respectively. The *Rdrs* column indicates the number of tasks reading-only this memory segment at a certain time. The *ww* flag (stands for a *writer waits*) indicates whether a task is waiting for previous readers to finish before it can run and write this memory segment. The latter case is known as the write-after-read hazards WAR. Although the WAR hazards and the write-after-write WAW hazards are false dependencies and are normally resolved using renaming techniques, Nexus++ supports them as a safe guard.

The *n_v*, *n_i*, and *p_i* columns stand for *next is valid* flag, *next entry index*, and *previous entry index* respectively, which builds up a linked-list structure inside the *Dependence Table* for entries that map to the same hash address. The *h_D* and *l_D* are the *has dummy* flag and *last dummy index* to implement the dummy entries mechanism explained in Section III-C in the *Dependence Table*, to overcome the limit on the number of tasks that may depend on a certain memory segment. These tasks are stored in the *Kick-Off List* of which is composed of the columns $T_1 \dots T_8$ of Table II.

Resolving new tasks dependencies: Every new submitted task to the *Task Maestro* is handled by the *Check Deps* block, of which pseudocode is shown in Listing 2. Listing 2 shows that for each entry *A* in the input/output list of the *newTask*, the *Dependence Table* is looked up, and an entry for *A* would be inserted if it was not found. On the other hand, if *A* was found, then an older task is already accessing it. In this case, the access modes are checked; if both the old and new tasks

access *A* as read-only, then the new task is granted access to *A*. However, if the older task is writing *A*, then the new task T_2 is added to the *Kick-Off List* of *A* as shown in Table II regardless of its access mode to *A* (RAW or WAW hazards when the new task T_2 is a reader or a writer of *A* respectively), and its *Dependence Counter* is incremented.

Finally, the WAR hazards are handled using the *ww* (a writer waits) flag in Table II. If a task T_1 is reading *B*, and T_{10} wants to write *B*, then T_{10} is added to the *Kick-Off List* of *B* as shown in Table II, its *Dependence Counter* is incremented, and the *ww* flag is set. Any other task that wishes to access *B*, regardless its access mode, will be added to the *Kick-Off List* of *B*, and its *Dependence Counter* is incremented.

After checking all inputs/outputs of a new task, the *Check Deps* block checks the new task's *Dependence Counter*, if it was 0, then the task does not depend on any other older tasks, and can be scheduled to run.

```
foreach A in parameters[newTask]
{
  if(A not exist){
    Add A to DT;
    if(newTask read-only A){
      DT[A].Rdrs=1;
      DT[A].isOut = false;
    }
    else
      DT[A].isOut = true;
  }
  else
  {
    if(newTask read-only A)
    {
      if(!DT[A].isOut && !DT[WriterWaits])
      {
        DT[A].Rdrs++;
      }
      else{
        DT[A].writeKickOffList(newTask);
        TP[newTask].DC++;
      }
    }
    else{
      DT[A].writeKickOffList(newTask);
      TP[newTask].DC++;
      if(!DT[A].isOut)
        DT[A].WriterWaits = true;
    }
  }
}
if(TP[newTask].DC == 0)
  GlobalReadyTasksList.write(newTask);
```

Listing 2. Checking dependencies for new tasks pseudocode

Handling finished tasks: Upon task completion, the *Handle Finished* block takes action. For example, for each entry *A* in the input/output list of the finished task T_1 , if T_1 has read-only *A*, then the *Rdrs* count of *A* is decremented. If it becomes 0 and no writer task is waiting (*ww* flag is false), then *A* is deleted from the *Dependence Table*. But if the *ww* flag was true, then a pending task T_2 must exist and is read from *Kick-Off List* of *A*.

On the other hand, if T_1 is a writer of *A*, and no tasks are waiting for *A*, then *A* is deleted from the *Dependence Table*. But if there are some tasks waiting for *A*, then the *Handle Finished* block will continuously read these tasks *IDs* one after the other as long as they read-only *A*, until it reads a task that is willing to write *A*, or the *Kick-Off List* of *A* is empty. Each time a reader is read from the *Kick-Off List*, the *Rdrs* count of *A* is incremented.

Dependency resolution in Nexus++ is more efficient than that in Nexus [10], since we use fewer and simpler tables and *Kick-Off Lists*. Nexus++ has only one table (*Dependence Table*) to maintain the task graph, and using the *Task Pool*'s

hAddr	v	fAddr	Size	isOut	Rdrs	ww	n_v	n_i	p_i	h_D	L_D	T _{ID}	...	T _{ID} or ptr_next Dummy
0xA	1	0x1A	4	1	0	0	0	-	-	0	-	T ₂	-	-
0xB	1	0x1B	4	0	1	1	0	-	-	0	-	T ₁₀	-	-
0xC	1	0x1C	4	1	0	0	1	111	-	1	333	T ₂₀	...	222
0x111	1	0x2C	4	1	0	0	0	-	C	0	-	T ₅₀	...	-
0x222	1	0x1C	4	-	-	-	-	-	-	1	333	T ₂₇	...	333
0x333	1	0x1C	4	-	-	-	-	-	-	0	-	T ₃₄	...	-

TABLE II

THE DEPENDENCE TABLE. (HADDR: HASH ADDRESS, FADDR: FULL ADDRESS, ISOUT: IS OUTPUT, RDRS: READERS COUNTER, N_V: NEXT IS VALID, N_I: NEXT ENTRY INDEX, P_I, PREV. ENTRY INDEX, H_D: HAS DUMMY ENTRIES, L_D: LAST DUMMY ENTRY INDEX, WW: A WRITER WAITS, T_x: TASK_x)

indices as task *IDs* eliminate the need to search the tables. In Nexus, on the other hand, three tables (containing two *Kick-Off Lists*) are used and are accessed always for all kinds of scenarios.

C. Dummy Tasks and Entries

In a *Task Descriptor*, a task has a limited number of inputs/outputs, so applications with tasks that have more inputs/outputs can not be executed directly on a system with Nexus. In addition, not all tasks necessarily have a number of inputs/outputs equal to the *Task Descriptor's* limit, which yields a poor memory utilization. We solve this problem by introducing dummy tasks. A dummy task will not be executed, it just takes the form of a task by having an entry in the *Task Pool*, only to store inputs/outputs that did not fit in the parent's input/output list. Figure 3 shows a scenario to demonstrate the need for dummy tasks. If T_x has $2n$ outputs, and a *Task Descriptor* can only store n of them (8 in our design), then dummy tasks (D_1 and D_2) are created having their inputs/outputs as those that did not fit in the parent's (T_x) *Task Descriptor*. A dummy task is simply a pointer that replaces the last entry of an input/output list.

In Table I, this mechanism is accomplished using the nD (number dummy) column along with the last column (P_8 or *ptr_next Dummy*) of a *Task Descriptor*. The number of the extra *Task Descriptors* needed is stored in the $nDummies$ column of the parent entry, as shown in the example in Table I. The *Task Descriptor* at index 98 has 10 inputs/outputs, which is more than maximum limit of 8 per *Task Descriptor*, that is why a new entry is occupied by this task, namely the *Task Descriptor* at index 99. The parent entry at index 98 has 1 in its $nDummies$ field, indicating that this task occupies in total 2 *Task Descriptors*, and the last entry in its input/output list now points to index 99. This process is done by the *Write TP* block.

Although this solves the problem of having a fixed, limited number of inputs/outputs per task, the maximum number of inputs/outputs is still bounded by the size of the *Task Pool*.

The same principle can be deployed in the *Dependence Table* shown in Table II, where the *Kick-Off List* has a limited size, thus restricting the number of tasks that might depend on a certain memory segment. As a solution, we add dummy entries to the *Dependence Table* to extend the *Kick-Off List* of a certain entry.

In Table II, a precise example is shown. Memory segment $0x1C$ is currently being written by a certain task T_1 , and the number of tasks that are waiting in the *Kick-Off List* of $0x1C$ doesn't fit in a single *Kick-Off List*. That is why the h_D flag

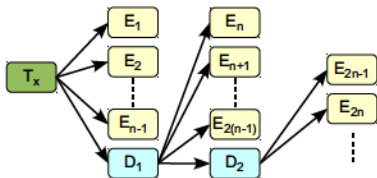


Fig. 3. Dummy Tasks/Entries added to the *Task Pool/Dependence Table*

of $0x1C$ is set, and the last entry in the *Kick-Off List* of $0x1C$ points to address 222, which contains also some tasks *IDs*, and also have another dummy entry at address 333 of the *Dependence Table*, where the rest of the waiting tasks reside.

Reading tasks *IDs* from the *Kick-Off List* of a certain memory address happens from the top of the first *Kick-Off List* of the chain. Whenever all tasks are read from the first *Kick-Off List*, this entry's data (except the *Kick-Off List* and the h_D fields) will be copied to the next dummy entry so that it becomes the new parent. For example, memory address $0x1C$ occupies 3 entries (at DT[0xC, 0x222, and 0x333]) in Table II. When all items in the *Kick-Off List* of DT[0xC] are read, this entry will be invalidated, and the parent entry of $0x1C$ will reside at DT[0x222]. This way, the *Dependence Table* is efficiently utilized, since DT[0xC] can now be reused by other memory segments, even before memory segment $0x1C$ is totally removed from the *Dependence Table*. This also allows direct (and hence, fast) access to the first *Kick-Off List*, since it always resides at the parent entry of a memory segment.

Dummy tasks are injected by the *Task Maestro* when needed at run time. They utilize memory well, and are scalable. The compiler could also add dummy tasks when it discovers that a task has more inputs/outputs than the maximum. However, the master core then would have to generate and submit more TDs, and [9] indicates that eventually the master core forms the bottleneck. Furthermore, the compiler can not add dummy entries to the *Dependence Table* since it depends on runtime information which is not available to the compiler. For these reasons we have decided that the *Task Maestro* adds dummy tasks and entries.

IV. EXPERIMENTAL SETUP

A. Benchmarks

Several benchmarks were used to evaluate Nexus++. First, we used a trace of parallel H.264 decoder decoding one full HD frame on a Cell Broadband Engine processor [11], consisting of 8160 tasks in total. The trace consists of tasks input/output information, tasks execution times and the time they have spent reading/writing their inputs/outputs from/to memory. On average a task spends $7.5\mu s$ for accessing off-chip memory and $11.8\mu s$ for execution [2]. The benchmark processes a matrix of 120×68 macroblocks and the dependency pattern is shown in Figure 4(a) [15]. Tasks are generated in serial execution order, which is from left to right and from top to bottom. Initially there is only one task ready for execution, but this number increases until halfway execution, after which it decreases again. This ramping effect influences the average amount of parallelism available in the benchmark and thus its scalability.

To evaluate Nexus++ for a range of dependency patterns, we created two additional synthetic benchmarks derived from the H.264 benchmark. Their dependency patterns are shown in Figure 4(b) and (c). We also used an additional benchmark without dependencies, i.e., has only independent tasks, in order to measure the maximum scalability of Nexus++. In contrast to dependency pattern (a), the dependency patterns

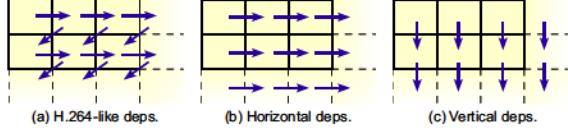


Fig. 4. Dependency patterns (120 × 68 blocks): (a) Ramp effect, (b, c) Fixed # of parallel tasks

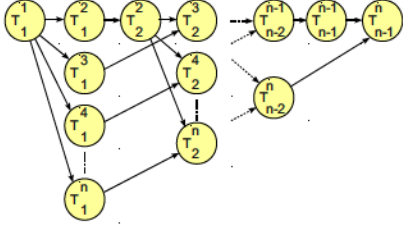


Fig. 5. Dependency pattern for the Gaussian elimination benchmark. T_i^j : i, j row and column numbers respectively

depicted in 4(b) and (c) do not suffer from the ramping effect. Instead, these dependency patterns provide a constant number of parallel tasks. In (b), however, the dependency pattern has the same direction as the order in which tasks are generated. As a consequence, the amount of effective available parallelism could be reduced by the speed of the addition process or the size of the *Task Pool*, since when the table is full, tasks of the first row have to be executed to make room for other tasks. Hence, leading to an indirect dependency.

To validate the dummy tasks/entries approach, the task graph of Gaussian elimination with partial pivoting [16] is used. In this benchmark, the number of tasks that depend on certain outputs depends on the size of the input matrix as depicted in the dependency pattern of Figure 5, assuming an $n \times n$ matrix.

The execution starts with one task (T_1^1), on which $n - 1$ tasks ($T_1^2 \dots T_1^n$) depend. After that only one task (T_2^2) can execute, and then $n - 2$ tasks, etc. Total number of tasks is relative to the matrix size, and equals $\frac{n^2+n-2}{2}$, where n is the matrix dimension. Each task performs number of floating point operations *FLOPs*. This number represent the weight W of a task and equals [16]:

$$W(T_i^j) = \begin{cases} n + 1 - i & \text{FLOPs if } i = j \\ n - i & \text{FLOPs if } i < j \end{cases} \quad (1)$$

where i, j are the row and column numbers respectively. Hence the duration of a task T_i^j equals $W(T_i^j)$, divided by the *GFLOPS* of one core. Each task also reads $W(T_i^j)$ floating point number from memory, and writes the same number back when finished.

Some tasks in the Gaussian elimination benchmark are really small (few *FLOPs*), but as can be seen in Formula (1) and Figure 5, the number of tasks of a certain weight is directly proportional to the weight itself. So the large portion of tasks are relatively coarse, and only a small portion are fine. Table III gives an overview about number and granularity of Gaussian tasks for different matrix sizes.

B. Simulation Environment

The Task Machine: Nexus++ was simulated using the *Task Machine*, a SystemC simulator of a task-based, trace-driven multicore system. The *Task Machine* is a fully configurable system that is designed to match modern real systems. Among the configurable parameters are the number of cores, core

Matrix dimension	# Tasks	Average task weight (FLOPs)
250	31374	167
500	125249	334
1000	500499	667
3000	4501499	2012
5000	12502499	3523

TABLE III
GAUSSIAN ELIMINATION TASKS FOR DIFFERENT MATRIX SIZES

System Parameter	Value
Cores clock freq.	2.0 GHz
Nexus++ clock freq.	500 MHz
On Chip Access Time	2 ns
Off Chip Access Time	12 ns
On chip bus bandwidth	2 GB/s
Memory bandwidth	10.67 GB/s
Task Descriptor (TD) size	78 Byte
Task Pool size	78 KB (1K TDs)
No. Parameters per TD	8
Dependence Table entry size	28 Byte
Dependence Table size	112 KB (4K entries)
Kick-Off list size	8 task IDs
TDs Sizes list size	1 KB
New Tasks list size	2 KB
TP Free Indices list size	2 KB
Global Ready Tasks list size	2 KB
Worker Cores IDs list size	2 KB
C_x RdyTasks list size	4 Bytes
C_x FinTasks list size	4 Bytes

TABLE IV
SYSTEM PARAMETERS

clock frequency, onchip/offchip memory access times, etc. Tasks information are read from experimental traces, which include tasks input/output information, and also their execution and memory access times. Thus task execution is simply modeled by waiting for a certain time. Memory accesses delays are modeled in the same way and memory contention is also modeled. The list of parameters and their values are shown in Table IV.

Nexus++ is simulated assuming a clock cycle time of 2 ns, which equals a clock frequency of 500 MHz. The *Task Maestro* tables and the FIFO lists are on-chip storage and therefore their access times are relatively fast. The hash table access time equals the on-chip access time multiplied by the number of lookups required per access.

The traces recording execution and communication times per task were generated after parallel H.264 decoding on a Cell processor [11]. Thus, the experiments are assuming a local-stores, shared-memory architecture. Nevertheless, Nexus++ concept can be applied to any other multicore architecture.

Design Space Exploration: The sizes of the *Task Maestro* tables and lists were empirically determined. They are summarized in Table IV. We observed, as will be shown later in Figure 6, that for the current benchmarks, the *Task Pool* should be able to contain 1k *Task Descriptors*. Assuming 8 parameters and a total 78 Bytes per task descriptor yields a *Task Pool* size of 78 KB. The *Dependence Table*, on the other hand, should be able to hold 4K entries, as will be shown later in Figure 6. Each entry size equals 28 bytes, which yields a table size of 112 KB.

Having 1K tasks in the *Task Pool*, 10 bits are needed index it and to identify a single task ID. This number is rounded up to multiples of a byte (i.e., 2 bytes), yields that 2KB are needed to store the IDs of a 1K tasks, which is the selected size for the *New Tasks list*, the *TP Free Indices list*, and the *Global Ready Tasks list*. 1 byte is allocated to store the size of one *Task Descriptor* upon its reception from the *Master Core*.

This gives a total size of 1KB for the *New Tasks list* to store the sizes of 1K *Task Descriptors*.

Simulating up to 512 worker cores, requires 9 bits to assign an individual ID to each core. Rounding this number up to multiples of bytes gives a 2KB *Worker Cores IDs list* size. Assuming double buffering, a worker core should be able to store two task IDs in its *RdyTasks* and *FinTasks* lists, which yields a size of 4 bytes per list.

Access Latencies: The access time for the ~ 100 KB on-chip memory structures (those are mainly the *Task Pool* and the *Dependence Table*) was determined using Cacti 5.3 [8], and was found to be 2 ns for each of them. Off-chip memory (RAM) access time is also determined using the same tool, and was found to be 12 ns per 128 bytes RAM chunk, assuming 32-bank 1GB of RAM, which is equivalent to a maximum memory bandwidth of 10.67 GB/s. The off-chip memory is assumed to have 32 banks, each having one read/write port. Therefore, no more than 32 tasks can access the memory at a given time, and this is how contention accessing off-chip memory is modeled.

The latency of preparation and submission of *Task Descriptors* by the master core was estimated. These times were measured in Nexus [9] in detail. As Nexus++ avoids off-chip communication in this part, we had to compensate for this. As a result, the task preparation was set to 30 ns, while the task submission is not fixed since it depends on the size of the input/output list of a task. The modeled on-chip bus is a very basic one. It is an 8-byte width bus, and its bandwidth is assumed to be 2GB/s which is a typical bandwidth of the state-of-the-art on-chip buses [13]. Every time the *Master Core* wishes to submit a task to the *Task Maestro*, it arranges the task's information into 8-byte words. The first word specifies the task's ID and function pointer, and every other word specifies a single parameter (including its address, size, and access mode). The *Master Core* also sends initially a handshaking word specifying the new task's number of words, and hence, number of its parameters. We assume that for each task submission, an initial (handshaking) bus delay of 5 cycles is needed, and each word takes 2 cycles (2GB/s bus bandwidth) to reach the *Task Maestro*. For example, a task with 4 parameters takes 10 cycles (20 ns), whereas an 8-parameters task takes 14 cycles (28 ns) submission delay.

V. EVALUATION

Nexus++ was tested under different conditions, varying the number of worker cores, the buffering depth, and with different dependency patterns.

Using double buffering, the independent tasks benchmark was performed varying the number of cores. Measuring the speedup against the single core experiment, the independent tasks benchmark achieved a speedup of $54\times$ on 64 cores. Furthermore, it achieved $143\times$ on 256 cores, assuming contention-free memory. When disabling task preparation delay, the resulting speedup was $221\times$ using 256 cores.

Design space exploration is also performed by running the independent tasks benchmark, on a 256-core system with double buffering, and contention-free memory. First, in order to determine the optimal *Dependence Table* size, all the other structures are configured to be very large, the *Task Pool*, for example, is configured to hold 8K *Task Descriptors* at once (given that the total number of tasks is 8160). The first column in Figure 6 shows the speedup achieved against varying the *Dependence Table* size, and fixing the *Task Pool* size at 8K entries. Maximum speedup equals $143\times$ when setting the *Dependence Table* size to 2K entries. However, the *Dependence Table* is set to 4K entries since this size

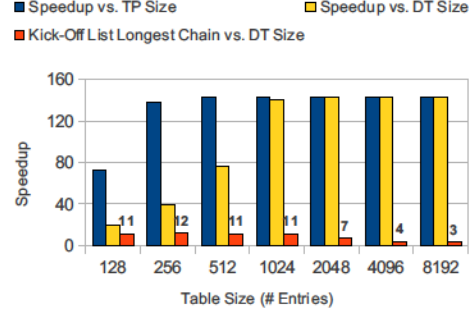


Fig. 6. Speedup achieved with varying the size of the *Task Pool* and fixing the size of the *Dependence Table* and vice versa. Also showing the effect of varying the *Dependence Table* size on the longest *Kick-Off List* chain.

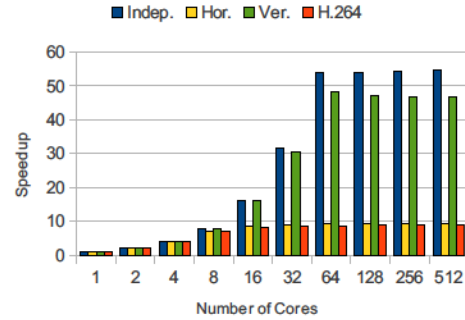


Fig. 7. Speedup achieved with different number of cores running tasks with dependencies shown in Figure 4.

enhances shorter *Kick-Off List* chain (almost half of that when the *Dependence Table* is set to 2K entries), as shown in the third column of Figure 6, as longer *Kick-Off List* chains implies a longer search time. The second column shows the speedup when varying the *Task Pool* size, and fixing the *Dependence Table* size at 8K entries. A *Task Pool* size of 512 entries is enough to achieve a speedup of $143\times$, however, a 1K entries *Task Pool* is chosen to allow a larger task window.

Figure 7 shows the achieved speedup for the benchmarks illustrated in Figure 4. As before, we simulate 8160 tasks with execution and communication times obtained from a parallel H.264 decoder [2]. The speedup is measured against the single core experiment of Nexus++ (double buffering enabled). Limited application scalability explains why the speedup gain decreases faster for the H.264 benchmark compared to the independent tasks speedup.

More interesting is the speedup gain difference between the benchmarks with horizontal and vertical dependencies illustrated in Figures 4(b) and 4(c), respectively. Although the *Task Pool* is larger than a single row, the processing of non-ready tasks before reaching the next ready task (first task in the second row of Figure 4(b)) limits the scalability of this benchmark to at most 8 cores, whereas the benchmark illustrated in Figure 4(c) scales well to 64 cores.

Figure 8 shows the speedup achieved by using different multicore systems to solve the Gaussian elimination problem (Figure 5) for different matrices of sizes ranging from 250×250 to 5000×5000 . Memory contention is modeled, and double buffering is used.

Although the size of the *Kick-Off List* of each of the *Dependence Table* entries is equal to 8, Nexus++ could handle the Gaussian elimination problem for matrices of large sizes. This is mainly because of the dummy entries added to the

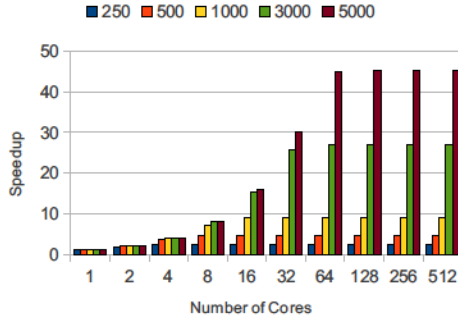


Fig. 8. Speedup achieved with different multicore systems running Gaussian elimination for different matrix sizes (legend shows matrix dimension)

Dependence Table. As shown in Figure 8, the matrix size has a great impact on the speedup gain and the scalability of the system, since a bigger matrix results in a larger number of tasks of larger granularity. A 5000×5000 matrix scaled up to 64 cores with a speedup factor of $45\times$. This experiment includes building and managing a task graph of 12502499 tasks with 3523 FLOPs per task on average as shown in Table III. Each single worker core is assumed to be able to do 2 GFLOPS, which means that the average computation time of each of the aforementioned tasks equals $1.77\mu s$.

Although the 250×250 has very small tasks ($83.5ns$ per task on average), Nexus++ could handle them. The benchmark scaled to 4 cores and a speedup of $2.3\times$ is achieved. This demonstrates the applicability of Nexus++ to any kind of applications, even those with very fine grained tasks.

All tables and FIFO lists in the Nexus++ task manager do not exceed 210KB of memory. Nevertheless, they are sufficient to perform all the objectives of Nexus++. The Task Superscalar [5], on the other hand, consumes more than 6.5MB and still has a static limit (19) on the number of inputs/outputs a task can have. Nexus++ introduces dummy tasks/entries in the *Task Pool* and the *Dependence Table* respectively, uses the *Task Pool* indices as tasks identifiers, and uses its internal structures more dynamically and efficiently, therefore tables sizes are relatively small.

VI. CONCLUSION

We have presented Nexus++, a hardware task management accelerator for the StarSs RTS. Compared to previous work Nexus++ makes four main contributions. First, it overcomes the limitation of Nexus that a task can only have a fixed, limited number of inputs/outputs by introducing dummy tasks in the *Task Pool*. It also overcomes the limitation that only a fixed, limited number of tasks can depend on a certain task by introducing dummy entries in the *Kick-Off Lists* of the *Dependence Table*. Second, it support double buffering by providing a task controller in each worker core. Third, it implements task dependency resolution more efficiently, since fewer hash table lookups are required to determine if tasks depend on each other. Fourth, we have presented a platform-independent implementation of Nexus++ whose parameters are fully configurable, while Nexus was integrated in a simulator of the Cell processor.

Experimental results obtained using a SystemC model show that double buffering achieved a speedup of $54 \times / 143 \times$ with/without modeling memory contention respectively, for a benchmark modeled after H.264 decoding. Furthermore, double buffering increases the scalability of the system. Eventually, for large (64 cores and more) systems, the speedup gain

starts to decrease, mainly because the application does not exhibit sufficient task-level parallelism, insufficient memory bandwidth, and/or because the master core cannot generate tasks fast enough to keep all worker cores busy. We have also shown that a benchmark modeled after Gaussian elimination, where the number of tasks that depend on a certain task is not constant, ran successfully and efficiently with an achieved speedup of $45 \times$ for an 5000×5000 matrix using 64 cores.

Although Nexus++ targets StarSs applications, parts of it can be reused for other programming models. For example, it contains hardware queues that can be used for low-latency retrieval of independent tasks. Future work will focus on how to make Nexus++ more versatile.

ACKNOWLEDGMENT

The research leading to these results has received funding from the European Community's Seventh Framework Programme [FP7/2007-2013] under the ENCORE Project (www.encore-project.eu), grant agreement n° 248647.

REFERENCES

- [1] G. Al-Kadi and A. S. Terechko. A Hardware Task Scheduler for Embedded Video Processing. In *Proc. 4th Int. Conf. on High Performance Embedded Architectures and Compilers*, 2009.
- [2] C. C. Chi, B. Juurlink, and C. Meenderinck. Evaluation of Parallel H.264 Decoding Strategies for the Cell Broadband Engine. In *Proc. 24th ACM Int. Conf. on Supercomputing*, 2010.
- [3] L. Dagum and R. Menon. OpenMP: an Industry Standard API for Shared-Memory Programming. *IEEE Computational Sci. Eng.*, 1998.
- [4] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proc. 6th Symp. on Operating Systems Design & Implementation*, 2004.
- [5] Y. Etsion, F. Cabarcas, A. Rico, A. Ramirez, R. M. Badia, E. Ayguade, J. Labarta, and M. Valero. Task Superscalar: An Out-of-Order Task Pipeline. *Microarchitecture, IEEE/ACM International Symposium on*, 0, 2010.
- [6] Y. Etsion, A. Ramirez, and R. M. B. Jesuslabarta. Cores as Functional Units: A Task-Based, Out-of-Order, Dataflow Pipeline. In *Proc. Int. Summer School on Advanced Computer Architecture and Compilation for Embedded Systems*, July 2009.
- [7] S. Kumar, C. J. Hughes, and A. Nguyen. Carbon: Architectural Support for Fine-Grained Parallelism on Chip Multiprocessors. In *Proc. 34th Annual Int. Symp. on Computer Architecture*, 2007.
- [8] H. Laboratories. Cacti 5.3. <http://www.hpl.hp.com/research/cacti/>.
- [9] C. Meenderinck. *Improving the Scalability of Multicore Systems, with a Focus on H.264 Video Decoding*. PhD thesis, Delft University of Technology, July 2010.
- [10] C. Meenderinck and B. Juurlink. A Case for Hardware Task Management Support for the StarSs Programming Model. In *Proc. 13th Euromicro Conf. on Digital System Design: Architectures, Methods and Tools*, 2010. Sp. Session on Multicore Systems: Des. and Apps.
- [11] D. Pham, S. Asano, M. Bolliger, M. Day, H. Hofstee, C. Johns, J. Kahle, A. Kameyama, J. Keaty, Y. Masubuchi, M. Riley, D. Shippy, D. Stasiak, M. Suzuoki, M. Wang, J. Warnock, S. Weitzel, D. Wendel, T. Yamazaki, and K. Yazawa. The design and implementation of a first-generation cell processor. In *Solid-State Circuits Conference, 2005. Digest of Technical Papers. ISSCC. 2005 IEEE International*, 2005.
- [12] J. Planas, R. M. Badia, E. Ayguade, and J. Labarta. Hierarchical Task-Based Programming With StarSs. *Int. J. High Perf. Comp. Appl.*, 2009.
- [13] Power.org. Power.org Embedded Bus Architecture Report. www.power.org/resources/downloads/Embedded_Bus_Arch_Report_1.0.pdf, 2008.
- [14] J. Reinders. *Intel Threading Building Blocks*. O'Reilly & Associates, Inc., 1st edition, 2007.
- [15] E. B. van der Tol, E. G. Jaspers, and R. H. Gelderblom. Mapping of H.264 Decoding on a Multiprocessor Architecture. In *Proc. SPIE Conf. on Image and Video Communications and Processing*, 2003.
- [16] M. Veldhorst. Gaussian Elimination with Partial Pivoting on an MIMD Computer. *Journal of Parallel and Distributed Computing*, 1989.